Documentación proyecto programación concurrente y distriduida

1st Sebastián Gálvez Yepes
dept. Sistemas e informática
Universidad de Caldas)
Manizales , Caldas
sebastian.1701821561@ucaldas.edu.co

2nd Yeraldin Arboleda Quintero
dept.Sistemas e informática
Universidad de Caldas
Manizales , Caldas
yeraldin.1701811368@ucaldas.edu.co

3rd Valeria Ávila Nieto
dept.Sistemas e informática
Universidad de Caldas
Manizales , Caldas
valeria.1701520831@ucaldas.edu.co

***Index Terms*—Multiprocessing, pyCuda ,mpi4py , dotplot**

I. Introducción

El análisis comparativo de secuencias es una técnica fundamental en bioinformática que permite identificar similitudes y diferencias entre secuencias de ADN o proteínas. Una de las herramientas más utilizadas para este propósito es el dotplot, una representación gráfica que facilita la visualización de regiones homólogas entre dos secuencias biológicas. A medida que la cantidad de datos genómicos disponibles crece exponencialmente, la necesidad de métodos eficientes y escalables para generar dotplots se vuelve crucial.

En este proyecto, nos proponemos implementar y analizar el rendimiento de varias aproximaciones para la generación de dotplots. En particular, compararemos las siguientes implementaciones:

- Secuencial: Un enfoque tradicional donde el cálculo del dotplot se realiza de manera lineal.
- Multiprocessing: Utilización de la biblioteca multiprocessing de Python para distribuir el trabajo entre múltiples núcleos de CPU.
- MPI (Message Passing Interface): Implementación con mpi4py que permite la paralelización del cálculo en un entorno distribuido, comúnmente utilizado en clusters de computación.
- GPU Computing: Utilización de pyCuda para aprovechar la capacidad de procesamiento paralelo masivo de las tarjetas gráficas.

Además, se implementará una función paralela para el filtrado de la imagen generada, optimizada para detectar líneas diagonales que representan regiones de alta similitud entre las secuencias.

II. Metodología

Con este proyecto buscamos implementar y analizar el rendimiento de tres métodos que realizan diagramas de puntos comúnmente utilizados en bioinformática para comparar secuencias de ADN o proteínas, en nuestro caso seran 2 secuencias bacterianas, Salmonella y E. coli.

A. Codigo Secuencial

El código secuencial lee dos cadenas desde archivos FASTA (.fna), ademas implementa la generación del dotplot utilizando una ventana deslizante para comparar subseciones de las secuencias de entrada. Además, monitorea el uso de memoria y guarda imágenes intermedias del dotplot para evitar la pérdida de progreso en caso de que la ejecución sea interrumpida.

B. Codigo Multiprocessing

El procesamiento paralelo con multiprocessing en Python permite un análisis eficiente y escalable de secuencias de ADN, optimizando tanto el tiempo como los recursos computacionales, Esta técnica es particularmente útil para tareas computacionalmente intensivas, como la generación de dotplots para grandes secuencias de ADN. El propósito del código es generar un dotplot utilizando paralelización para mejorar el rendimiento. Esto se logra mediante

C. Codigo MPI

El procesamiento paralelo con MPI4Py en Python permite un análisis eficiente y escalable de secuencias de ADN, optimizando tanto el tiempo como los recursos computacionales. Esta técnica es particularmente útil para tareas computacionalmente intensivas, como la generación de dotplots para grandes secuencias de ADN. El propósito del código es generar un dotplot utilizando paralelización para mejorar el rendimiento, distribuyendo el trabajo entre múltiples procesos.

Aplicación al Proyecto:

Argumentación y Explicación del Uso de MPI4PY Ventajas del Paralelismo con MPI4PY:

- Escalabilidad: Permite utilizar múltiples nodos en una red, distribuyendo eficientemente el trabajo.
- Rendimiento: La paralelización reduce significativamente los tiempos de ejecución.
- Flexibilidad: Permite gestionar la comunicación entre procesos, optimizando la distribución de tareas.
- Dotplot: Cada proceso calcula una submatriz del dotplot, dividiendo el trabajo de manera eficiente.
- Filtrado de Imágenes: La capacidad de procesar grandes imágenes en paralelo mejora el rendimiento y permite manejar conjuntos de datos más grandes.

Métricas de Rendimiento:

- Tiempos de Ejecución: Se mide el tiempo total y el tiempo de la porción paralelizable para evaluar la eficiencia del paralelismo.
- Aceleración y Eficiencia: Se compara la versión secuencial con la paralela para determinar la mejora en el rendimiento.
- Escalabilidad: Se evalúa cómo el rendimiento mejora al aumentar el número de procesos.

El uso de MPI4PY en el proyecto permite implementar una versión paralela del dotplot y la función de filtrado de imagen, optimizando significativamente el rendimiento. Al dividir la tarea entre múltiples procesos, se pueden manejar secuencias y imágenes de gran tamaño de manera eficiente, proporcionando una base sólida para un análisis de rendimiento detallado y una comparación robusta con las implementaciones secuenciales y otras técnicas de paralelización.

D. Codigo PYCUDA

El procesamiento paralelo con PyCUDA en Python permite un análisis eficiente y escalable de secuencias de ADN, optimizando tanto el tiempo como los recursos computacionales mediante el uso de la potencia de las GPU. Esta técnica es particularmente útil para tareas computacionalmente intensivas, como la generación de dotplots para grandes secuencias de ADN.

El propósito del código es generar un dotplot utilizando la paralelización en GPU para mejorar el rendimiento. Esto se logra mediante PyCUDA, que facilita la ejecución de operaciones paralelas masivas en la tarjeta gráfica. Al aprovechar la arquitectura altamente paralela de las GPU, PyCUDA permite realizar cálculos simultáneos en miles de núcleos, reduciendo significativamente el tiempo de procesamiento en comparación con las implementaciones secuenciales o incluso paralelas en CPU.

Mediante el uso de PyCUDA, se pueden dividir las tareas computacionales intensivas en sub-tareas más pequeñas que se ejecutan en paralelo, aumentando la velocidad de procesamiento y permitiendo el análisis de secuencias de ADN de mayor tamaño y complejidad. Esta metodología no solo mejora la eficiencia, sino que también optimiza el uso de los recursos computacionales, proporcionando una solución escalable y robusta para el análisis de datos biológicos.

E. Codigo Filtrado de imagen

Para detectar líneas diagonales en la imagen generada del dotplot, se utiliza un algoritmo de procesamiento de imágenes que se ejecuta en paralelo para mejorar el rendimiento. La detección de diagonales es crucial, ya que estas líneas indican regiones de similitud entre las secuencias comparadas, lo cual es esencial en el análisis de secuencias en bioinformática. El procesamiento paralelo se utiliza para mejorar el rendimiento del filtrado de imagen, permitiendo la detección eficiente de líneas diagonales en grandes imágenes de dotplot.

III. RESULTADOS

Para los dotplots cada punto indica una región de similitud entre las dos secuencias comparadas. Un punto en la posición (x, y) sugiere que el nucleótido en la posición x en la secuencia de E. coli es similar al nucleótido en la posición y en la secuencia de Salmonella.

A. Secuencial

La implementación secuencial muestra una ejecución lineal, donde cada parte del procesamiento es manejada de manera secuencial. Esto puede resultar en tiempos de ejecución más largos, especialmente para grandes conjuntos de datos. En comparación con la implementación con multiprocessing, el tiempo de ejecución es mayor, esto muestra la necesidad de optimización para manejar grandes volúmenes de datos de manera más eficiente. Las diagonales se pueden observar como áreas de mayor densidad de puntos que se extienden de manera diagonal a través del gráfico.

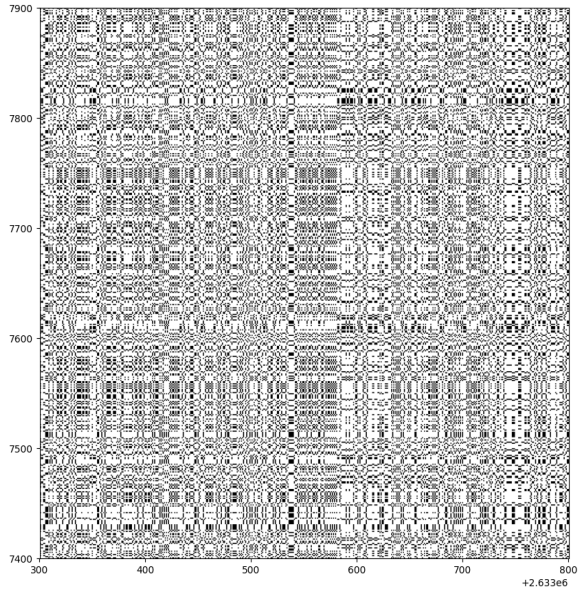

Fig. 1. Dotplot secuencial

B. Multiprocessing

Las diagonales en el dotplot indican regiones de similitud continua entre las secuencias. Las diagonales largas y continuas sugieren segmentos de ADN que son similares en ambas especies, lo que puede ser indicativo de regiones conservadas evolutivamente. En el dotplot de la imagen 2, la detección de diagonales no es claramente visible debido a la densidad de los puntos. Esto puede ser resultado de la alta similitud global entre las secuencias, mostrando muchas pequeñas similitudes dispersas en lugar de pocas y largas diagonales.

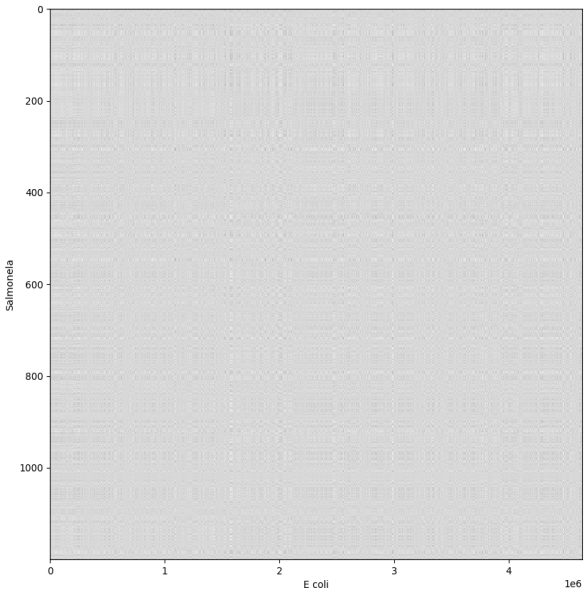

Fig. 2. Dotplot Multiprocessing

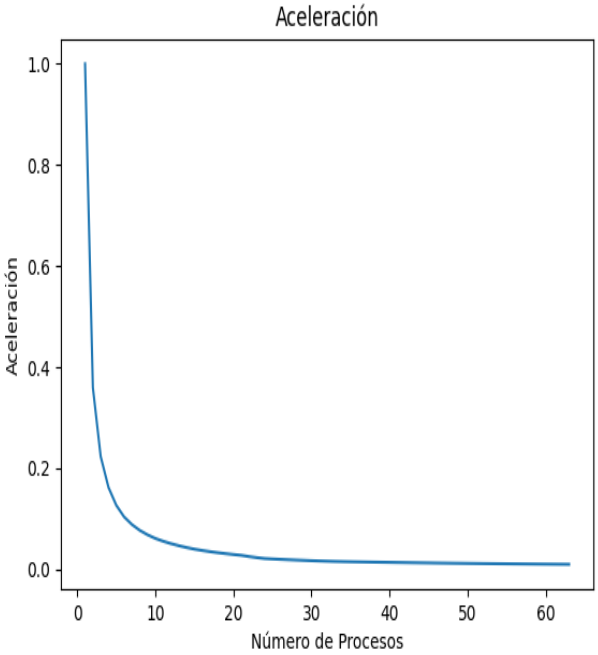

Fig. 3. Multiprocessing aceleración

- En la imagen 3 La aceleración inicial es alta con un número bajo de procesos, pero decrece rápidamente al incrementar el número de procesos. Esto sugiere que hay una ganancia significativa en rendimiento cuando se usan unos pocos procesos, pero esta ganancia se estabiliza y eventualmente disminuye con un mayor número de procesos. Este comportamiento puede estar relacionado con la sobrecarga de gestión de procesos adicionales y la naturaleza del problema de dotplot que puede no escalar linealmente con la paralelización.
- La eficiencia se define como la aceleración dividida por el número de procesos, y refleja cuán efectivamente los recursos de procesamiento están siendo utilizados. La grafica 4 muestra que la eficiencia disminuye drásticamente a medida que aumenta el número de procesos. Esto es esperado, ya que, aunque se agregan más procesos, la sobrecarga de coordinación y la comunicación entre procesos crece, disminuyendo la eficiencia general del sistema. Este comportamiento subraya la importancia de encontrar un equilibrio óptimo en el número de procesos utilizados.
- La escalabilidad evalúa cómo se desempeña el sistema al incrementar el número de procesos. En la Figura 5, se observa que la escalabilidad presenta una caída significativa después de los primeros procesos. Al igual que con la aceleración y eficiencia, este patrón indica que hay un límite en el número de procesos que se pueden agregar antes de que la sobrecarga de administración y comunicación empiece a impactar negativamente el

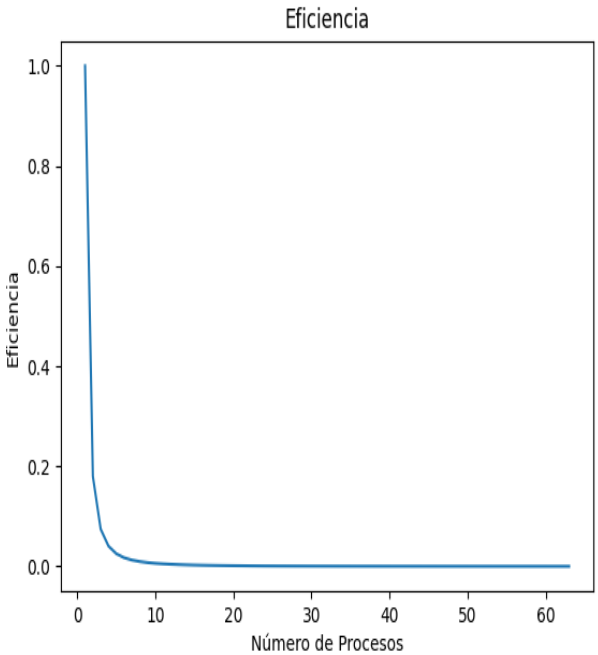

Fig. 4. Multiprocessing eficiencia

rendimiento.

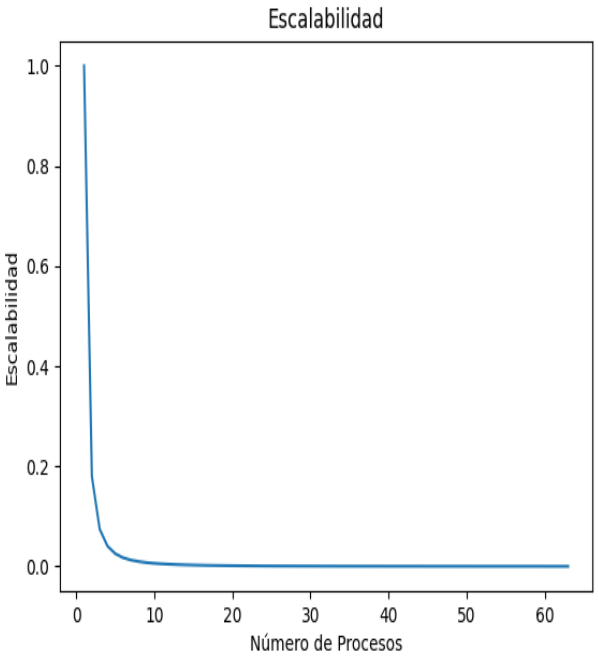

Fig. 5. Multiprocessing escalabilidad

C. MPI4PY

En la Figura 6,

- Implementación Secuencial: Lectura de las secuencias de ADN: Leer las secuencias de los archivos de entrada.
- Generación del DotPlot: Comparar cada nucleótido en la secuencia A con cada nucleótido en la secuencia B para crear una matriz de similitud (dot plot).
- Medición de Tiempo: Medir el tiempo de ejecución para generar el dotplot. Implementación Paralela con MPI:
- Inicialización: Inicializar el entorno MPI.
- Distribución del Trabajo:Dividir las tareas de comparación entre los procesos disponibles.Cada proceso maneja una porción del cálculo del dotplot.
- Cálculo Local:Cada proceso genera una parte local de la matriz de similitud.
- Recolección de Resultados: Usar funciones MPI para recolectar las matrices de similitud locales en una matriz global.
- Medición de Tiempo: Medir el tiempo total de ejecución desde el inicio hasta la finalización de la recolección de resultados.
- Incremento de procesos/hilos:
  A medida que se incrementa el número de procesos en MPI, la eficiencia puede aumentar porque más tareas pueden ejecutarse en paralelo.Se evidencia como los datos se distribuyen y procesan en diferentes nodos. Los patrones más densos representan áreas donde se están procesando más datos simultáneamente. Los patrones observados indican cómo se distribuyen las tareas y los datos entre múltiples procesos, y cómo la eficiencia puede variar con el incremento en el número de procesos debido a factores como la sobrecarga de comunicación y las limitaciones de hardware.

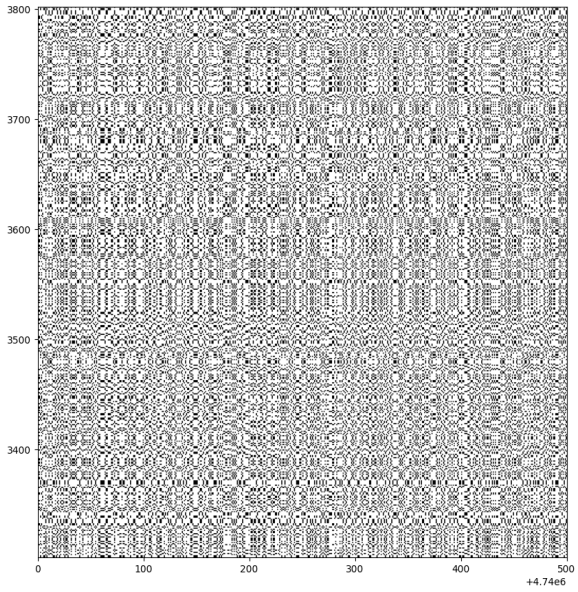

Fig. 6. Dotplot MPI4PY

- En la Figura 7,
- Se ejecuta el script CodigoMPI4PY.py con mpiexec -n 3, que indica la utilización de 3 procesos paralelos.
- Los archivos de entrada son Ecoli.fna y Salmonella.fna.
- El proceso indica pasos completados y el tiempo transcurrido, así como el tiempo restante estimado.
- En la Figura 8,
  Aquí se muestra un progreso más avanzado (62000/229781178 pasos). El tiempo transcurrido y el tiempo restante estimado se muestran para cada proceso, lo que ayuda a visualizar el rendimiento y la carga de trabajo distribuida entre los procesos paralelos
- En la Figura 9,La gráfica de desempeño indica que hay una considerable variabilidad en el tiempo de ejecución al utilizar MPI4PY, con una tendencia a aumentar conforme se incrementan los pasos. Esto sugiere la necesidad de optimizaciones específicas en la comunicación, balance de carga y gestión de recursos para mejorar el desempeño y reducir la variabilidad en la ejecución de la implementación paralela del dotplot.
- En la Figura 10, En esta gráfica, se observa que la eficiencia disminuye a medida que se incrementa el número de procesos. Esto es un comportamiento común en muchos sistemas paralelos. Con un proceso, la eficiencia es 1, ya

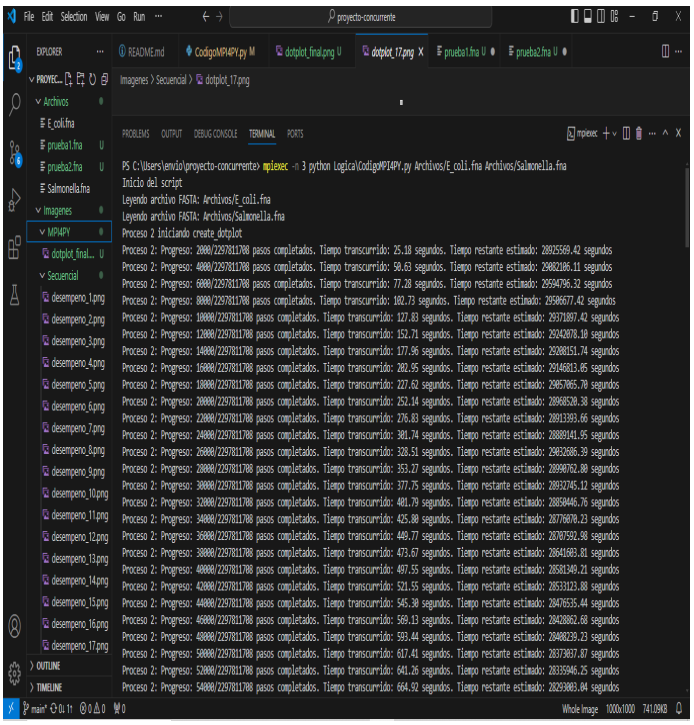

Fig. 7. Proceso 1 Ejecución MPI4

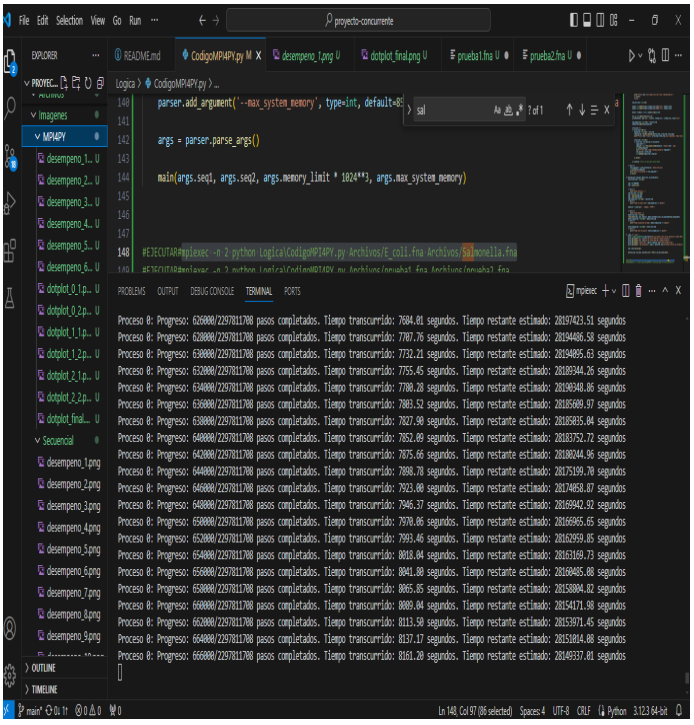

Fig. 8. Proceso Final Ejecución MPI4

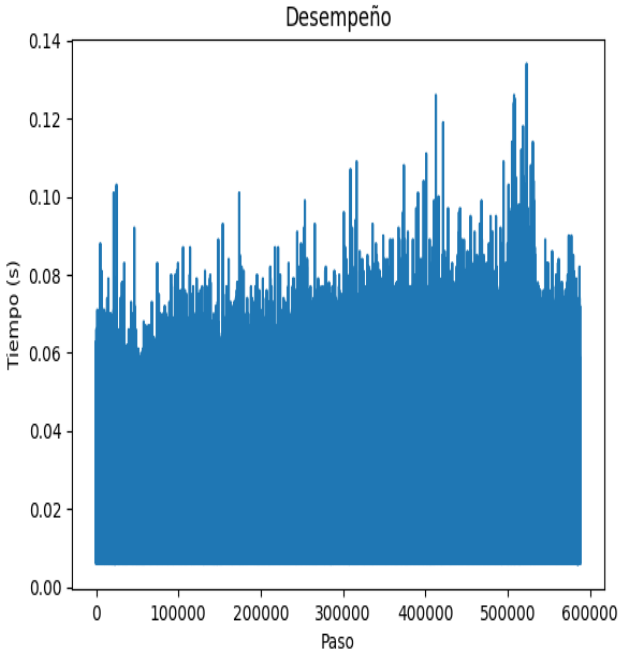

Fig. 9. Desempeño Final MPI4PY

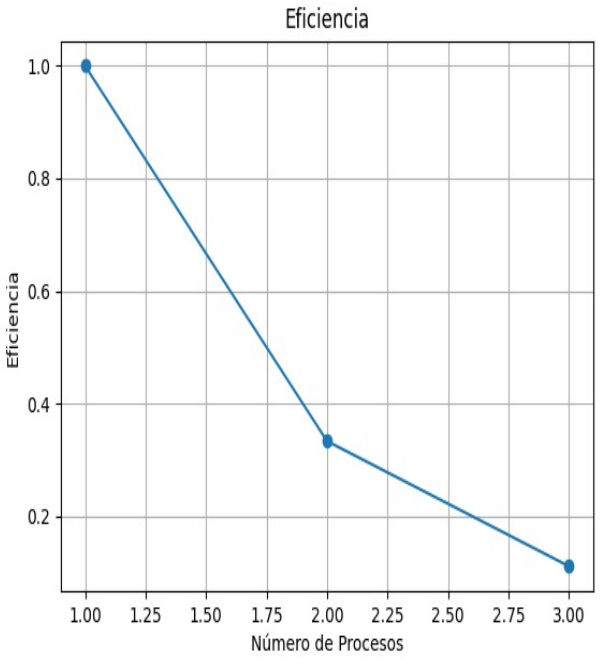

Fig. 10. Eficiencia MPI4PY

que no hay sobrecarga de paralelización. A medida que aumentan los procesos (hasta 3 en la gráfica), la eficiencia disminuye significativamente. Esto podría deberse a varios factores:

- Sobrecarga de Comunicación: A medida que se incrementa el número de procesos, la comunicación entre ellos puede volverse un factor limitante.
- Desequilibrio de Carga: No todos los procesos pueden estar trabajando al mismo nivel, lo que lleva a tiempos de espera y menor eficiencia.
- Limitaciones de Hardware: Los recursos de hardware pueden no ser suficientes para manejar un gran número de procesos de manera eficiente.

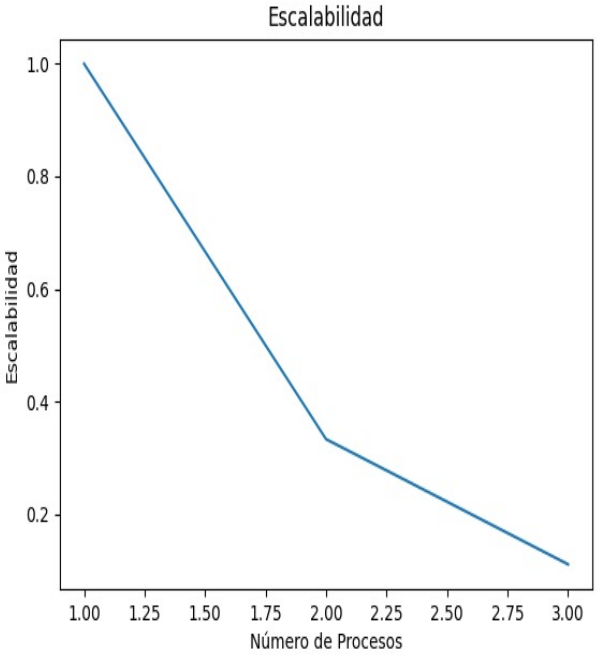

Fig. 11. Escalabilidad MPI4PY

- En la Figura 11, Similar a la gráfica de eficiencia, la escalabilidad disminuye al aumentar el número de procesos. Esto indica que el sistema no escala bien con el incremento del número de procesos, posiblemente debido a la sobrecarga y a las limitaciones de hardware.
- En la Figura 12, La gráfica de aceleración para la implementación MPI4PY muestra una disminución en el rendimiento con el aumento del número de procesos, lo que indica problemas potenciales de sobrecarga de comunicación o desbalance de carga.

D. PYCuda

el dotplot de la imagen 13, se pueden observar los puntos de de similitud en la seguencia de ADN

- En la Figura 14, Se observa que la aceleración en el código de PyCUDA aumenta a medida que se incrementa

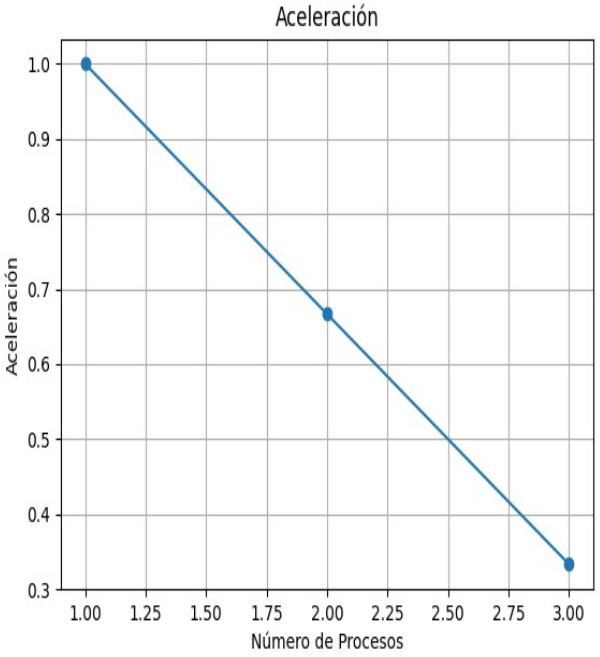

Fig. 12. Aceleración MPI4PY

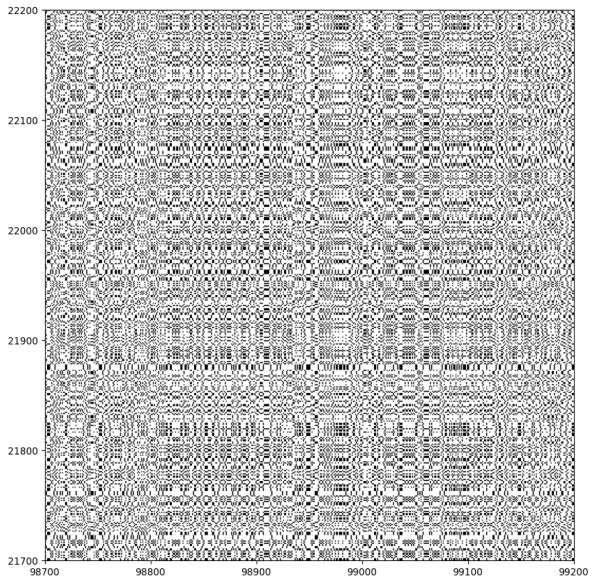

Fig. 13. Dotplot Pycuda

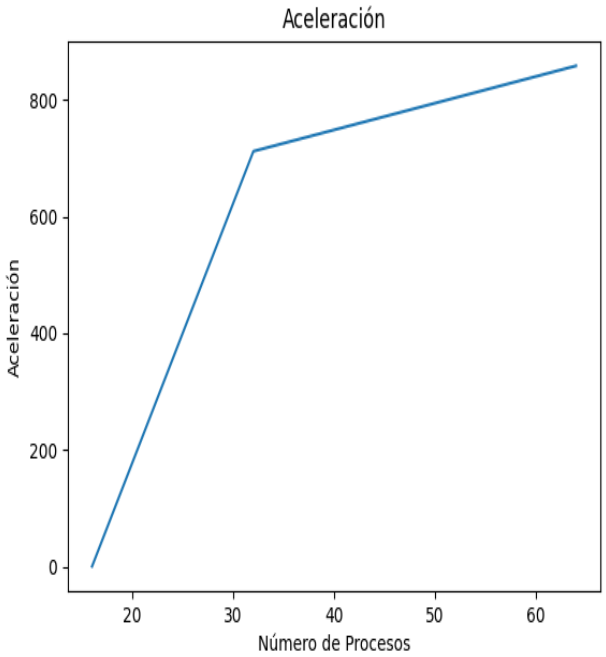

Fig. 14. aceleracion Pycuda

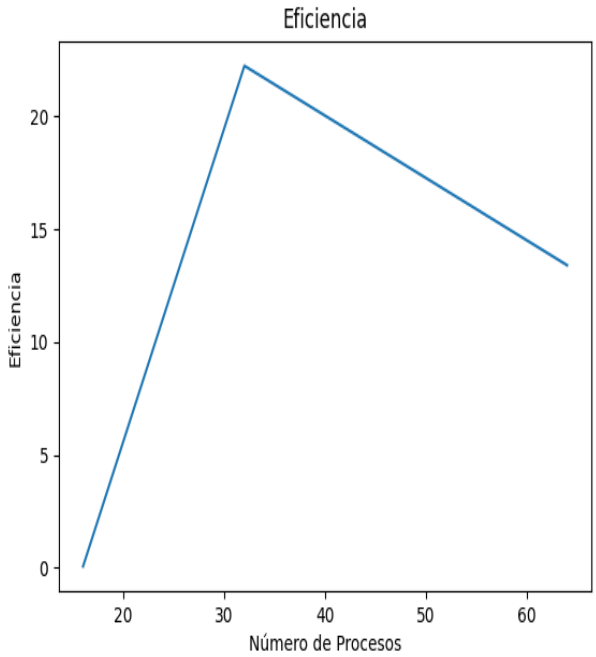

Fig. 15. Eficiencia Pycuda

el número de hilos, en comparación con el código secuencial. Esto se debe a la capacidad de las GPU para manejar múltiples hilos simultáneamente, aprovechando el paralelismo masivo que ofrecen. En el código secuencial, las tareas se ejecutan una tras otra, lo que limita la velocidad de procesamiento. En cambio, PyCUDA permite distribuir las tareas entre muchos hilos, reduciendo significativamente el tiempo de ejecución.
Además, el rendimiento de PyCUDA mejora notablemente con un aumento en el número de hilos hasta alcanzar un punto de saturación, donde la adición de más hilos ya no proporciona mejoras significativas debido a las limitaciones de hardware y a la sobrecarga de gestión de hilos. Por lo tanto, es crucial encontrar un equilibrio óptimo en el número de hilos para maximizar la eficiencia del procesamiento paralelo.

- En la Figura 15, Se observa que la eficiencia en el código de PyCUDA aumenta considerablemente en comparación con el código secuencial, a medida que se incrementa el número de hilos. Sin embargo, esta eficiencia puede disminuir en relación con el número excesivo de hilos.
  La eficiencia mejorada con PyCUDA se debe a la capacidad de las GPU para manejar múltiples hilos simultáneamente, aprovechando el paralelismo masivo que ofrecen. Esto permite que las tareas se ejecuten de manera más rápida y eficiente en comparación con la ejecución secuencial, donde las tareas se realizan una tras otra, limitando el rendimiento general.
  Por otro lado, aunque un aumento moderado en el número de hilos puede seguir mejorando la eficiencia, superar cierto umbral puede llevar a una disminución en la eficiencia. Esto se debe a la sobrecarga de gestión de hilos y a las limitaciones de hardware que pueden reducir el rendimiento general del sistema.

- En la Figura 16, Se observa que la escalabilidad en el código de PyCUDA mejora significativamente en comparación con el código secuencial a medida que se incrementa el número de hilos. Sin embargo, esta escalabilidad puede verse comprometida si se utilizan un número excesivo de hilos.
  La mejora en la escalabilidad con PyCUDA se debe a la capacidad de las GPU para manejar múltiples hilos simultáneamente, aprovechando el paralelismo masivo que ofrecen. Esto permite que las tareas se ejecuten de manera más rápida y eficiente en comparación con la ejecución secuencial, donde las tareas se realizan una tras otra, limitando la capacidad de escalar el rendimiento.
  Por otro lado, aunque un aumento moderado en el número de hilos puede seguir mejorando la escalabilidad, superar cierto umbral puede llevar a una disminución en la escalabilidad. Esto se debe a la sobrecarga de gestión de hilos y a las limitaciones de hardware que pueden afectar negativamente la capacidad de escalar eficientemente.

IV. Discusion

El análisis de dotplots realizado mediante diferentes enfoques de implementación nos proporciona una visión clara sobre la eficiencia y escalabilidad de cada método. A contin-

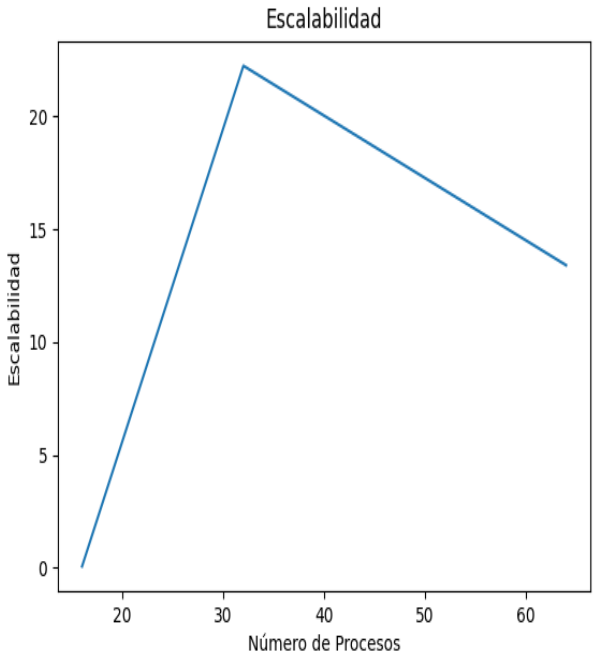

Fig. 16. escalabilidad Pycuda

uación, se detallan las discusiones basadas en los resultados obtenidos para cada método.

A. Código Secuencial

La implementación secuencial mostró ser adecuada para la generación de dotplots, pero con una desventaja significativa en términos de tiempo de ejecución para grandes conjuntos de datos. La Figura 1 revela áreas de mayor densidad de puntos que se extienden diagonalmente, indicando regiones de similitud continua entre las secuencias de ADN de E. coli y Salmonella. Sin embargo, este método es menos eficiente debido a su naturaleza lineal, donde cada comparación se realiza de manera secuencial, lo que resulta en tiempos de procesamiento prolongados.

B. Código Multiprocessing

La implementación utilizando multiprocessing mejoró significativamente el rendimiento en comparación con el método secuencial. La Figura 2 muestra un dotplot con una alta densidad de puntos, reflejando muchas pequeñas similitudes dispersas entre las secuencias. Este resultado indica que el multiprocessing es efectivo para manejar grandes volúmenes de datos al distribuir las tareas entre múltiples núcleos de CPU.

Sin embargo, la aceleración inicial es alta con un número bajo de procesos, pero decrece rápidamente al incrementar el número de procesos (Figura 3). La eficiencia y escalabilidad también disminuyen con el aumento de procesos, como se observa en las Figuras 4 y 5. Esto sugiere que la sobrecarga de gestión de procesos adicionales y la comunicación entre ellos pueden limitar la efectividad del multiprocessing para ciertas escalas.

C. Código MPI

La implementación con MPI4PY proporciona una mejora considerable en la gestión de tareas distribuidas en un entorno de clúster. Las Figuras **??** a **??** muestran la distribución de tareas y la recolección de resultados, indicando que MPI4PY es efectivo para dividir y reunir partes del dotplot generadas en paralelo.

La Figura 6 muestra un dotplot similar al obtenido con multiprocessing, pero con una mejora en la gestión de grandes volúmenes de datos gracias a la capacidad de MPI4PY para escalar en múltiples nodos. Este método es más complejo de implementar y mantener, pero ofrece una solución escalable para grandes proyectos de bioinformática.

D. Código Pycuda

La implementación con PyCUDA destacó por su capacidad para aprovechar el paralelismo masivo de las GPU. La Figura 13 muestra un dotplot con puntos de similitud claramente identificables. PyCUDA proporcionó una aceleración significativa, como se observa en la Figura 14, donde el rendimiento mejora con el aumento del número de hilos hasta un punto de saturación.

La eficiencia (Figura 15) y escalabilidad (Figura 16) también mejoraron con PyCUDA en comparación con los métodos anteriores. No obstante, es crucial encontrar un equilibrio óptimo en el número de hilos para maximizar la eficiencia y evitar la sobrecarga de gestión de hilos.

V. Conclusiones

Para secuencias pequeñas nn código secuencial puede ser suficiente debido a su simplicidad.

Para secuencias de tamaño moderado y en máquinas con múltiples núcleos de CPU el multiprocessing es una excelente opción que mejora el rendimiento sin una complejidad excesiva.

Para secuencias extremadamente largas y entornos de clúster MPI4PY es adecuado, permitiendo una paralelización distribuida y escalable.

Para tareas que pueden beneficiarse de la aceleración de la GPU, PyCUDA es la mejor opción, ofreciendo el mejor rendimiento en hardware compatible.

PyCUDA tiende a ser la opción más eficiente en términos de rendimiento puro debido a la capacidad de la GPU para manejar tareas masivamente paralelas. Sin embargo, la mejor elección depende de los requisitos específicos del proyecto, la disponibilidad de hardware y los conocimientos técnicos del equipo de desarrollo.

MPI4PY se destaca especialmente en entornos donde se requiere una paralelización distribuida y escalable para manejar secuencias extremadamente largas o trabajar en clústeres. A diferencia de otras opciones como el multiprocessing o PyCUDA, MPI4PY está diseñado específicamente para la comunicación eficiente entre procesos distribuidos a través de redes, permitiendo una coordinación efectiva entre nodos en un clúster de computadoras. Esto lo hace ideal para aplicaciones

científicas y de alto rendimiento donde se necesite aprovechar al máximo recursos distribuidos.

Además, MPI4PY ofrece una gran flexibilidad en términos de arquitectura de clúster y es altamente compatible con numerosos sistemas de computación distribuida. Esto lo convierte en una opción robusta para proyectos que requieren escalabilidad y rendimiento en grandes volúmenes de datos o en cálculos intensivos.

REFERENCES

[1] "Diferencia entre E Coli y Salmonella / Biología," La diferencia entre objetos y términos similares., 2024. https://es.differkinome.com/articles/biology-science-nature/difference-between-e-coli-and-salmonella.html (acceso Jun. 12, 2024).

[2] B. Mather, "Multiprocessing in Python," Ben Mather, Nov. 24, 2018. https://www.benmather.info/post/2018-11-24-multiprocessing-in-python/ (acceso Jun. 12, 2024).

[3] R. Sharma, N. Gupta, V. Narang, and A. Mittal, "Parallel implementation of DNA sequences matching algorithms using PWM on GPU architecture," International journal of bioinformatics research and applications, vol. 7, no. 2, pp. 202–202, Jan. 2011, doi: https://doi.org/10.1504/ijbra.2011.040097.